EMERGENT REQUIREMENTS FOR SUPPORTING INTRODUCTORY PROGRAMMING

NATALIE J COULL
School of Computing and Engineering Systems, University of Abertay Dundee,
Bell St., Dundee, DD1 1HG
(+44 1382 30 8938)
n.coull@abertay.ac.uk

Ishbel M M Duncan
School of Computer Science,
University of St Andrews
North Haugh, St Andrews, KY16 9SX
(+44 1334 46 3270)
Ishbel@cs.st-andrews.ac.uk

ABSTRACT

The problems associated with learning and teaching first year University Computer Science (CS1) programming classes are summarized showing that various support tools and techniques have been developed and evaluated. From this review of applicable support the paper derives ten requirements that a support tool should have in order to improve CS1 student success rate with respect to learning and understanding.

Keywords

Introductory programming, learning, design, human factors, theory.

1. INTRODUCTION

Students learning to program are often initially frustrated by not only having to learn code syntax, but by having deeper issues of understanding than is evident from any compilation failure. Students require to engage with programming on both a cognitive and a conceptual level and, as help is not always immediately at hand in a busy laboratory or in private development, an automated tool supporting those concepts, as well as fundamental syntax and semantics, would be a useful addition to their educational armoury. The work described here is the first part of a doctoral research project to determine what was necessary in a student support tool for learning to program, and, secondly, to build and evaluate such a tool. The aim of the initial research, therefore, was to review and then synthesize current research and practice in the field and to determine fundamental support requirements that any learning system should have.

The paper outlines the current state of the art in learning to program research and practice followed by a discussion of the applicable learner models. These give rise to the emergent requirements for a support tool and the conceptual architecture for such a tool is then outlined. The paper concludes with a short summary of the second part of the research project in which the requirements were substantiated within a prototype tool.

2. RELATED RESEARCH

The specific task of learning to program attracts particular challenges, and it is well recognized as a time-consuming and frustrating process for the majority of novices (Johnson 1990; Edwards 2003; Kelleher and Pausch 2005; Bennedsen and Caspersen 2007; Eckerdal 2009). Progress has been made in terms of identifying the underlying difficulties faced by novices by investigations of possible underlying influences (Simon, Fincher et al. 2006) and, separately, direct comparison with expert programmers (Mead, Gray et al. 2006; Kay 2007). Some older studies, for example, Soloway and Ehrlich (1984) explored the difference between novice programmers and experts. These older studies lay the foundation of our current understanding of the programming process. The Soloway and Ehrich paper states that the process of programming requires at least two different skill sets: program comprehension (i.e. the ability to understand each program part) and, secondly, plan recognition (i.e. the ability to understand the interactions among each program part). Program comprehension is dependent on an understanding of the form and meaning of each symbol and each line, and this skill is necessarily evident in experts. This study also demonstrates that understanding arises as a result of recognition of the interaction among program lines in the form of a

systematic plan. This plan recognition is evidently present in experts but not in novices. Similarly, Mayer (1981) considers the differences between novices and experts and presents a useful analogy to programming in terms of playing chess. Mayer discusses a study where subjects were presented with chessboard configurations and then asked to reconstruct them from memory. Although chess masters performed much better than less experienced players when the chessboards came from real games, there was little difference when the chessboards contained randomly placed pieces. This demonstrates that the chess masters do not have better memories than the less experienced players, but an understanding of the patterns that can exist among pieces on the board. Mayer compares this to a study by Shneiderman (1980) where expert programmers were able to recall more lines of code than novices in a meaningful program but performed no better than novices when the program contained random lines. Mayer suggests that it could be useful to teach novices to program using "chunks" or "schemas" to help them learn the patterns evident in programming.

More recently, Kay (2007) suggests using errors to determine and then ultimately predict erroneous actions of beginners, intermediates and then advanced programmers. He also states, along with Ebrahimi and Schweikert (2006) that mistakes came from incorrect plans or mental models and a previous experiment (Lazonder and Van Der Meij 1995) demonstrates that novices require context specific information when an error has occurred. Ebrahimi (1994) undertakes a review of existing work on the difficulties faced by novice programmers and clearly states that the two challenges noted by the studies above impact significantly on the learning process for novices. It is clear that knowledge of the syntax and underlying semantics is required to develop programs. However, Ebrahimi states unambiguously that program plans are also a fundamental requirement to successful program development. Ebrahimi also shows, through observational study, that the concepts of "language constructs" and "plan composition" are central to programmer development and intimately linked. Ebrahimi concludes that programmers *cannot form plans without language and cannot use language without plans* and that, consequently, these must be taught concurrently.

The literature described above demonstrates that there are two challenges in learning to program. The first, here termed *program formulation* is the syntax and semantics of the individual program parts that are fundamental to the formulation of any program. The second, here termed *problem formulation* is therefore the challenge of identifying and structuring the necessary program components to formulate a programmed solution to a specific problem.

Both Fuller et al. (2007) and Rountree et al. (2005) state that code comprehension and programming are two separate skills and need to be developed. Lister et al. (2004) investigated students' ability to trace programs. All these skills are considered essential for developing a student along the path towards experience. Malan and Halland (2004) identified pitfalls to avoid when teaching programming, specifically identifying the problem of teaching being too abstract and confusing the novice student. Edwards states that students need "explicit, continually reinforced practice in hypothesizing about the behaviour of their programs….[and] need frequent, useful and immediate feedback" (2003). Students may believe that once their program compiles then it works and must be correct. This suggests that their mental model is perhaps not accurate or that they have failed to understand the logic of their code. Murphy et al. (2008) states that novices behave unproductively when debugging their programs implying confused mental models. Norris et al. (2008) notes that better performing students spend less time on a task, using fewer compilations and writing less code implying better mental models of their code.

Lahtinen et al. (2005) suggest that studying alone and example programs were the best mechanism for learning and Fetaji et al. (2007) states that learning to program in a classroom approach is not the most appropriate strategy. Hassinen and Mayru (2006) support a learning-by-doing theory and Shaffer (2003) further suggests limiting the number of concepts introduced in a class. Further, Pillay (2003) states that one on one tutoring is the best mechanism for helping novice programmers.

3. LEARNER MODELS

In order to improve the learning process that underpins the development of expertise in programming, a number of models of student learning have been devised (Mayer 1981; Mayer 1997; McGill and Volet 1997). These models are typically formulated in terms of stages in progressing from a novice to an expert programmer and can be used to identify and describe the composite aspects of the activity of learning to program. In turn, these activities, and the accordant skill base, can be used to determine the direction that we expect students will follow and provide guidance and structure on how we need to help students begin this process of learning.

Lund (2002) recognizes the utility of the work by Mayer (1997) in particular and has proposed a unified framework of programming expertise to underpin the development of a teaching support tool, which clearly defines the categories of knowledge required to program. These categories are identified, as per Mayer, as Syntactic, Semantic, Schematic and Strategic. These levels can form the foundations of the learning support that should be offered to students and can be mapped explicitly on to the two challenges presented above, program formulation and problem formulation. The existing description of each challenge is further refined by the use of Mayer's learning model;

Program Formulation
Program Formulation encapsulates both the Syntactic and Semantic knowledge levels of Mayer (1997). According to Lund (2002), Syntactic knowledge is specifically concerned with the syntax, or form, of a programming language and the ability to apply that knowledge. Correct syntactic expression of constructs within a particular programming language allows development of Semantic knowledge, i.e. the operation of those individual constructs, which will enable them to develop a mental model of program execution or "what goes on inside the computer as a result of a line of code" (Lund 2002). This mental model, combined with knowledge of the underlying syntax, represents the skill set needed for program formulation.

Problem Formulation
Lund also defines Schematic knowledge as the ability to recognize patterns in code developed for previous problems, also known as programming plans, and apply these plans to form a solution to the current problem. As noted earlier, novice programmers have difficulty recognizing these patterns and, even when the patterns are made explicit in one context, they are unable to transport these patterns to a given problem, i.e. another context. The final layer in the model, Strategic, may be defined as the techniques employed to create and monitor plans, so following a software development process (Lund 2002). Therefore, knowledge at the syntactic and semantic levels maps directly on to program formulation and knowledge at the schematic and strategic levels maps directly on to problem formulation.

Simon et al. (2006), state that students with pre-existing strengths in strategic or algorithmic articulation are invariably successful programmers. One could interpret this as these students progress quickly through the levels because they are already thinking at the problem formulation level and can therefore cope with lower level issues such as syntax errors. Kordacki (2007) implies the ability of learners to use representation systems to express their problem-solving strategies will make for better learners, suggesting the use of both program and problem formulation. Soloway and Ehrlich (1984) previously claimed that a major source of code problems came from plan composition failures, i.e. program formulation. A similar claim was also made by Winslow (1996) when he stated that "..novice programmers know the syntax and the semantics of individual statements, but they do not know how to combine these features into valid programs."

4. EMERGENT REQUIREMENTS

From reviews of existing tools (Kelleher and Pausch 2005; Coull 2008) it is evident that a support tool may be used to facilitate the learning process of novices who are learning to program. Further, from this review, a series of requirements emerge. Generally, there is diversity in methods and approaches used to teach the complex, interleaved concepts associated with programming. Therefore, a support tool that is sufficiently flexible to accommodate many different instructional and delivery designs is of the most value. Ideally, the support tool should be decoupled from any specific development requirement and further operate as an add-on to any chosen programming environment. No support tool that aids program formulation with respect to syntax should promote dependence on that tool to such an extent that the student is not able to progress beyond the remit of that tool. To remove this dependence, and to benefit from support that takes account of the taught context, it is necessary to present the student with both the standard compiler messages and any supplementary error messages enhanced with context-relevant support. Again at the program formulation level, and in recognition of the frequent semantic errors which students make, a support tool should detect these common mistakes and provide relevant warnings to the novice. To further reduce the risk of dependency the level of support should be reduced over time. In addition to these steps, the use of a support tool should be voluntary as this affords students the opportunity to direct their own learning and allows those students with existing experience to opt out of extended support provision and develop programs more quickly.

To support a student more fully in the development of a solution to a given problem, i.e. problem formulation, a support tool needs to have knowledge of both the key constructs and the relationship among those constructs necessary for a working solution. A support tool should be able to offer assistance to the student in identifying which constructs are needed to solve a particular problem and make sure that their solution meets

the requirements of the exercise. This in itself raises two issues. First, it is important that a support tool does not direct a student towards a single model solution, but allows the student to develop a correct solution of their own design, i.e. the support tool must recognize that variant solutions exist and these should be supported. Second, the tool must provide support in stages, guiding the student through the development process, rather than directing the student towards a complete solution from the outset or only operating with a near solution. Finally, to reinforce concepts already taught to the students, a support tool should be able to direct novices toward relevant, context sensitive (i.e. the general taught context such as terminology and bespoke teaching packages) and context aware (i.e. the specific teaching activity currently undertaken) teaching materials. The feedback provided, including links to teaching materials, should provide an effective platform upon which to base student-tutor dialogue.

In summary, Figure 1 below lists the requirements of a support tool:

1	Present both standard compiler and enhanced support concurrently
2	Link to teaching resources as a means of information delivery and student-tutor dialogue
3	Identify and advise on commonly observed semantic errors
4	Embody knowledge of key constructs needed to solve a given problem
5	Embody knowledge of the relationships between the constructs needed to solve a problem
6	The knowledge should be disseminated to students in successive stages
7	Where appropriate, ensure that this knowledge accommodates variant solution forms
8	Provide support for different problems
9	All forms of support may be progressively reduced over the teaching period
10	Use of the tool must be voluntary on the part of the student

Figure 1 Core Requirements of an Effective Support Tool

5. A Conceptual Architecture for Learning Support Tools

After identifying all of the requirements identified above, the next phase of the research was to develop the architecture for support tool development. The architecture identifies the components needed in a system to develop a technology-based tool that supports novices learning to program. Figure 2 illustrates those components and the interrelationships among them. Central to the architecture is a software implementation that draws on established resources and integrates the different aspects of support provided into a single interface. These aspects are outlined here to provide an overall context and are then explored more fully in subsequent sections.

The architecture comprises five main dimensions: Syntactic, Semantic, Schematic and Strategic structure support together with interaction, i.e. invocation and feedback, with the support tool itself. To provide Syntactic Support for program formulation it is necessary to supplement the (original) error messages arising from the compilation process with extended text, where this text is sensitive to the taught context. To provide Semantic Support for program formulation it is necessary to perform checks on the program code for common (logic) errors and provide warnings to students, which indicate these logic errors. Knowledge of the problem domain (a specific practical exercise) may be used to inform the development of strategic and schematic checks for solution formulation, i.e. checks for key constructs and the relations among those key constructs respectively. Through this mix of syntax, semantic, schematic and strategic checks, the support tool may provide feedback to the student. This support must be integrated into the current development environment to provide a seamless interface that is available whenever students elect to use it.

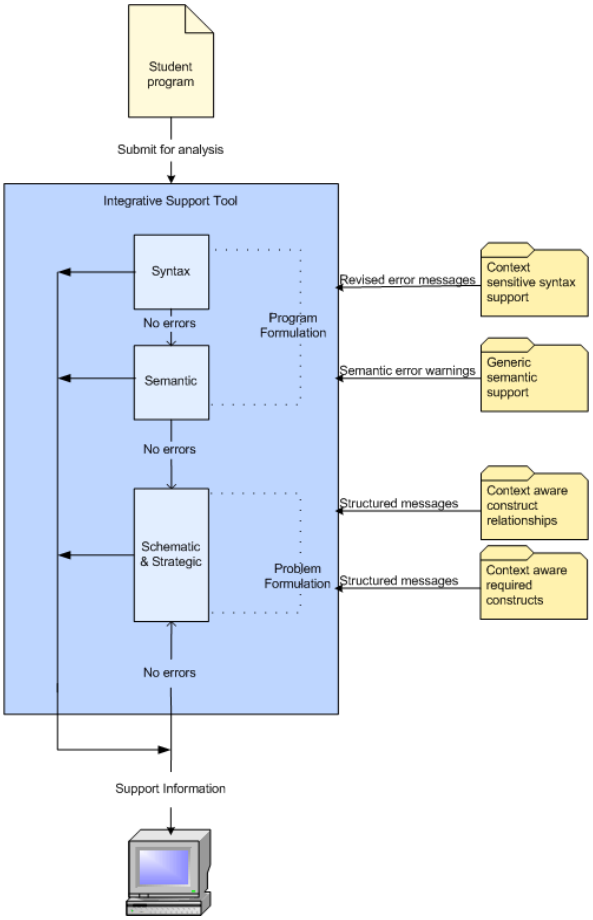

Figure 2 Generic Architecture

Program Formulation Support

Supporting the student in writing syntactically correct code is imperative for program formulation. As discussed earlier, difficulties with syntax can affect student learning of both the language and the underlying concepts. Many universities use industry-standard programming languages as a tool for teaching introductory programming. The compiler error messages produced by these programming languages are designed for expert programmers, and so their successful interpretation may depend on a degree of knowledge not possessed by a novice. Rather than *replace* these expert level error messages, these messages should be *extended* with content more suitable for novices (Requirement 1). Since the student should always be presented with the original error message, they should soon recognize with the aid of the extended messages why these errors have occurred and be able to resolve compiler-only errors in due course. These extensions should use dialogue that the student is familiar with and, where possible, relate to the content of the actual module (Requirement 2).

Similarly, Semantic Support is also very important for program formulation. Although problems at this level do not occur with the same frequency as syntax errors, their presence is less obvious to the novice, and so they are harder to detect, since they are manifest only at run-time. As there is no compiler support for these errors students may spend a long time trying to identify and then rectify the problem. Indeed, based on observations (Jadud 2006) students often make structural changes to their program in an attempt to correct its behaviour rather than identifying a (more trivial) semantic error. Some other support tools, for example BlueJ (Johnson 2006), recognize the value of providing support for semantic problems and have incorporated a series of semantic checks into the software support. To address Requirement 3 a support tool should perform checks on syntactically valid code for commonly observed semantic errors. Again, the dialogue used in feedback should be something that the student is familiar with and, where possible, relate to the content of the taught material (Requirement 2).

Problem Formulation Support

To successfully formulate any program a novice must have sufficient knowledge of the syntactic and semantic aspects of the language. However, the formulation of a solution to a given programming problem requires identification of the required structure of the program, i.e. the interrelationships among the component parts of the program. Many students need assistance with identifying such a program schematic (Lahtinen, Ala-Mukta et al. 2005). A support tool that has knowledge of the appropriate constructs (Requirement 4) and these interrelationships (Requirement 5) required for a specific programming exercise can be used to fully guide the student towards creating a solution. Rather than present the student with a list of all the necessary constructs and interrelationships, students should be guided in a stepwise manner through a solution development (Requirement 6). This guide may then exploit the schematic patterns inherent in introductory exercises to promote a systematic approach to software development. Perhaps the most fundamental and yet conceptually challenging level of learning occurs at the strategic level. Here, a support tool should check that the basic components required to solve the problem are in place (Requirement 4). Problems in identifying this basic set of components are driven by a failure to recognize the (often implied) functional requirements stated in a question and how those requirements are translated into the program development process. A support tool should make clear, again in a stepwise manner (Requirement 6), that translation by making transparent the links between program requirements and the components needed to meet those requirements. Further, both here and in the Schematic Support offered, the support tool must recognize that there may be more than one possible solution for a given problem and provide support for a range of solutions appropriate to the problem posed (Requirement 7). Finally the tool should provide support for different types of problems (Requirement 8), ideally the range of problems covered in their course.

Feedback

To enable students to become more confident in their abilities, and develop a skill set independent of any additional support, it is necessary to reduce the amount of feedback provided as teaching progresses (Requirement 9). The progressive reduction in support over the taught period will ultimately eliminate dependence on a support tool. To ensure that the feedback is relevant and contains terminology that the student is familiar with, the feedback should make a direct relation to core and/or supplementary teaching material where possible (Requirement 2). To direct their own learning students must have the opportunity to work with just the basic toolset, for example the compiler messages, and so invocation of any software support must be at the individual student's discretion (Requirement 10).

6. EXPERIMENTAL EVIDENCE

To evaluate the derived requirements and the generic architecture, a prototype implementation, SNOOPIE, was built and linked to specific programming courses. The remit and feedback of the tool support was initially informed by an extended series of laboratory observations. These demonstrated the multi-faceted nature of problems that students encounter whilst they are learning to program and how these problems can be mapped onto the different levels of program and problem formulation. Detecting erroneous problem formulation is not easy to identify and analysis was done through interview, questionnaires and experimental evidence. Erroneous program formulation can be detected through compilation errors and the number of attempts and time taken to achieve a solution. The support tool was fully evaluated and demonstrated to have a significant impact on the learning process for novice programmers. From a series of experiments with multiple student cohorts over two years in two universities each using a different series of laboratory practicals and lectures, five hypotheses were evaluated: that program formulation support improved short-term student performance, that problem formulation support improved short-term student performance, that problem formulation support reduced the time taken to achieve an ideal solution, that a combination of problem and program formulation improved long-term student performance and that the system was perceived as having a positive impact on the student learning process. Hypotheses 1 to 3 were tested using laboratory experiments, hypothesis 4 was analysed using student performance at the module level combined with interviews with staff and students and finally, hypothesis 5 was tested via experiments, questionnaires and interviews with students. The results demonstrated that, using the SNOOPIE tool:

1. Students receiving program formulation support were able to solve errors faster than those without support, i.e. program formulation support improves short-term performance
2. Students receiving problem formulation support were more able to produce a solution that met the criteria than those without support, i.e. problem formulation support improves short-term performance
3. Students receiving problem formulation support took longer to produce a solution than those without support. It was concluded that students were not being "spoon-fed" solutions and took time to think in

detail about a problem and consequently problem formulation support does not reduce the time taken to achieve an ideal solution.

4. Student confidence was seen to improve through the use of the support tool and although the results were less significant than those in earlier hypotheses, student long-term performance with both program and problem formulation support was better than those with only program formulation support.
5. Students perceived the support tool as adding value and having a positive impact on their learning experience.

7. SUMMARY

Learning to program is recognized as a complex task that novices find challenging. Existing literature demonstrates that learning to program is difficult because of the need to learn the rules and operation of the language (program formulation), and the concurrent need to interpret problems and recognize the required components for that problem (problem formulation). There exist many endeavours to support the novice in this activity, including software tools that aim to provide a more supportive environment than that provided by standard software facilities. This paper presents a new methodology for developing learning support tools that addresses the dual task of program and problem formulation. A review of existing teaching tools that support the novice programmer lead to a set of requirements for a support tool that encompasses the processes of both program and problem formulation. The set of requirements is encapsulated in a generic architecture for software tool development. A prototype support system was then built from the architecture and a series of hypotheses was developed to determine system validity. By using a combination of evaluation methods including anecdotal evidence, interviews, questionnaires and experiments, the results demonstrated how a series of automated analyses at different stages in the student's development of a solution improved both student confidence and performance.

8. REFERENCES

Bennedsen, J. and M. E. Caspersen (2007). "Failure Rates in Introductory Programming." Inroads (SIGSCE Bulletin) **39**(2): 33-36.

Coull, N. J. (2008). Support for Novices in an Object Oriented Programming Integrated Environment. School of Computer Science. St Andrews, Scotland, KY16 9SX, University of St Andrews. **PhD**.

Ebrahimi, A. (1994). "Novice programmer errors: language constructs and plan composition." International Journal of Human-Computer Studies **41**: 457-480.

Ebrahimi, A. and C. Schweikert (2006). "Empirical Study of Novice Programming with Plans and Objects." Inroads (SIGSCE Bulletin) **38**(4): 52-54.

Eckerdal, A. (2009). Novice programming students' learning of concepts and practice. Information Technology. Upsalla, Sweden, University of Upsalla. **Ph.D.**

Edwards, S. H. (2003). Rethinking Computer Science Education from a Test-First Perspective. OOPSLA'03 148-155.

Fetaji, M., S. Loskovska, et al. (2007). Combining Virtual Learning Environment and Integrated Development Environment to Enhance eLearning. International Conference on Information Technology Interfaces. Croatia.

Fuller, U., C. G. Johnson, et al. (2007). "Developing a Computer Science-specific Learning Taxonomy." SIGCSE Bulletin **39**(4).

Hassinen, M. and H. Mayra (2006). Learning Programming by Programming: A Case Study. Baltic Sea '06 Koli Calling.

Jadud, M. C. (2006). Methods and Tools for Exploring Novice Compilation Behaviour. Proceedings of the 2006 International Workshop on Computing Education Research. Canterbury, UK, ACM**:** 73-84.

Johnson, C. (2006). Abstract interpretation of student programs as a strategy for courseware development. In Methods, Materials and Tools for Programming Education, Tampere Polytechnic**:** 14-19.

Johnson, W. L. (1990). Understanding and debugging novice programs, Artificial intelligence and learning environments. Scituate, MA, Bradford Company.

Kay, R. H. (2007). "The role of errors in learning computer software." Computer & Education **49**: 441-459.

Kelleher, C. and R. Pausch (2005). "Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers." ACM Computing Surveys **37**(2): 83-137.

Kordacki, M. (2007). Modeling and multiple representation systems in the design of a computer environment for the learning of programming and C by beginners. World COnference on e-learning in corporate, government, healthcare and higher education (e-learn 2007). T. Bastiens and S. Carliner. Quebec, Canada, Chesapeacke, VA:AACE**:** 1634-1641.

Lahtinen, E., K. Ala-Mukta, et al. (2005). A Study of the Difficulties of Novice Programmers. ITiCSE'05 10th Annual SIGCSE Conference on innovation and Technology in Computer Science Education. Caparica, Portugal, ACM Press**:** 14-18.

Lazonder, A. W. and H. Van Der Meij (1995). "Error-information in tutorial documentation: supporting users' errors to facilitate initial skill learning." International Journal of Computer Studies **42**: 185-206.

Lister, R., E. S. Adams, et al. (2004). A Multi-national study of reading and tracing skills in novice programmers. ITiCSE WGR04: Working group reports from ITiCSE on Innovation and technology in Computer Science Education.

Lund, G. (2002). Aspects of the Program Development Process used by Learner Programmers School of Computing. Dundee, Scotland, University of Abertay. **Ph.D.**

Malan, K. and K. Halland (2004). Examples that do Harm in Learning Programming. OOPLSLA'04. Vancouver**:** 83-87.

Mayer, R. E. (1981). "The Psychology of How Novices Learn Computer Programming." ACM Computing Surveys (CSUR) **13**(1): 121-141.

Mayer, R. E. (1997). From Novice to Expert. Handbook of Human Computer Interaction 2nd.Edition. M. Helander, T. K. Landauer and P. Prabhu, Elsevier-Science B.V.

McGill, T. J. and S. E. Volet (1997). "A Conceptual Framework for Analysing Students' Knowledge of Programming." Journal of Research on Computers in Education **29**(3): 276-297.

Mead, J., S. Gray, et al. (2006). A Cognitive Approach to Identifying Measureable Milestones for Programming Skill Acquisition. ITiCSE 2006. Bologna**:** 182-193.

Murphy, L., G. Lewandowski, et al. (2008). Debugging: the good, the bad, and the quirky -- a qualitative analysis of novices' strategies. 29th SIGCSE Technical Symposii, on Computer Science. Protland, Oregon**:** 163-167.

Norris, C., F. Barry, et al. (2008). ClockIT: Collecting quantitative data on how beginning software developers really work. 13th Conference on Innovation and Technology in Computer Science Education. Madrid, Spain**:** 37-41.

Pillay, N. (2003). "Developing Intelligent Programming Tutors for Novice Programmers." Inroads (SIGCSE Bulletin) **35**(2): 78-82.

Rountree, J., N. Rountree, et al. (2005). Observations of Student Competency in a CS1 Course. Australasian Computing Education Conference 2005, Conferences in Research and Practice in Information Technology. **42**.

Shaffer, D., W. Doube, et al. (2003). Applying Cognitive Load Theory to Computer Science Education. Workshop of the Psychology of Programming Interest Group.

Simon, S. Fincher, et al. (2006). Predictors of Success in a First Programming Course. Eighth Australian Computing Education Conference (ACE2006).

Soloway, E. and K. Ehrlich (1984). "Empirical Studies of Programming Knowledge." IEEE Trans. on Soft. Eng **SE-10**(5): 595-609.

Winslow, L. E. (1996). "Programming Pedagogy – a Psychological Overview." SIGCSE Bulletin **28**: 17-22.